1.0 INTRODUCTION

1.0 INTRODUCTION

The Quick Code Generator (QCG) generates object code and emits the loader tables for the compiler when an optimization level of LOW or DEBUG is selected. QCG will also handle MEDIUM optimization when implemented.

There is already a QCG for Fortran/VE V1. Because of the additional requirements for version 2 of the compiler and changes in Front End table structures, it will be necessary to redesign and rewrite the first half (QCG proper); nevertheless, the old QCG design will be kept where there is no need to make alterations. The other half, the assembler, is relatively unaffected as all it does is generate the loader tables.

There are two major parts to this document. First is a brief overview of the current QCG design. The other describes how QCG V2 will differ.

1.1 SUMMARY_OF_REQUIREMENTS

In addition to what is supported in version 1, the following features are to be supported in version 2:

* Allocated and assumed-shape arrays.
* Array sections and vector expressions.
* User array-valued functions.
* Inline expansion of selected intrinsic math routines (OL=LOW/MEDIUM).
* New statements: WHERE, ALLOCATE, DEALLOCATE, and WHERE blocks.
* New array-valued intrinsics: SUM, COUNT, ANY, DOTPRODUCT, MATMUL, RANK, PACK, UNPACK, SEQ, SHAPE, SIZE, LBOUND, UBOUND, ALLOCATED, PRODUCT, MAXVAL, and MINVAL.
* Inline range checking code (subscripts and substrings).
* Inline character assignment and comparison code.

Hooks must be provided for the following features for painless implementation in future releases:

* Integer*n, Logical*n, Real*n, and Complex*n support.
* MEDIUM optimization level.

1.0 INTRODUCTION
1.1 SUMMARY OF REQUIREMENTS

Other requirements:

* Must generate vector instructions for "simple" explicit vector expressions.

* Must support Target Mainframe (TM) parameter.

* Must support (Professional) Programming Environment.

* OL=LOW/DEBUG compilation performance is critical.

1.2 OPTIMIZATIONS

The only optimization performed for OL=LOW is basic block squeezing, the elimination of redundant arithmetic operations, address calculations, and loads and stores within the confines of a basic block. After squeezing comes dead code elimination. This is what is done in FTN/VE V1 QCG.

Basic block squeezing and dead code elimination will also be the only optimization perform for OL=DEBUG. The difference from OL=LOW is that a basic block always terminates at end of statement.

2.0 GLOSSARY

Front End (FE)	The scanner, parser, and semantic analyser parts of the compiler.
IL	The main representation of a FORTRAN program in QCG for V2. It is generated by turple translation, may undergo optimizations, and is then translated into prebinary.
I-number	The ordinal of a pseudocode or IL instruction. Usually used to refer to the result of an instruction. I-numbers are unique to a basic block for IL, and are unique to the entire program unit for pseudocode.
Prebinary	The final representation of a FORTRAN program in QCG just prior to the generation of loader tables. It is virtually machine code.
Pseudocode	The main representation of a FORTRAN program in QCG for V1. It is generated by turple translation and is translated into prebinary.
Turples	What a FORTRAN program becomes when the FE gets through with it -- a very high-level intermediate language.

3.0 FTN/VE V1 QCG DESIGN OVERVIEW

3.0 FTN/VE V1 QCG DESIGN OVERVIEW

There are three passes in the current QCG. The first pass translates turples into pseudocode. Each turple will generally produce zero or more pseudocode instructions, and possibly add some entries to certain tables. The second pass translates each pseudocode instruction into zero or more prebinary instructions. Finally, the third pass generates loader tables from the prebinary instructions and other QCG and Front End tables. In all three cases, the translation process proceeds sequentially through the tables from top to bottom.

3.1 TURPLE TRANSLATION

There are two groups of turples, each of which are processed in a different fashion. The first group are translated via macro expansion using code skeletons, generally the same skeletons used by the Front End for constant expression reduction. The other group is handled by a gigantic CASE statement. Generally, code skeletons are used except where table entries are added or consulted, which they cannot do. Turples for I/O or subroutine calls fall into this category.

During turple translation, the following are performed (and all at the same time!):

* Variables and arrays are assigned addresses as they are referenced for the first time.
* Complex arithmetic is expanded into real arithmetic.
* Certain generic intrinsics are expanded.
* Basic block boundaries are detected.
* Redundant instructions are squeezed.
* Dead code is eliminated.
* Definition-Use chains are created -- but only within the boundaries of a basic block.
* Address calculations are expanded.
* Numerous tables, such as CILIST, IOLIST, subroutine arguments, constant tables, and others, are built.

3.0 FTN/VE V1 QCG DESIGN OVERVIEW
3.1 TURPLE TRANSLATION

* Machine-dependent peephole optimizations are performed, such as deciding between MULX and MULXQ, or ADDX and ADDXQ, when an operand is a constant.

3.2 PSEUDOCODE FORMAT

The pseudocode is very close to machine code, having almost a one-to-one correspondence between pseudocode operations and machine code operations. The differences are that pseudocode uses an infinite register model, three-address arithmetic instructions instead of two-address format, variable addresses have yet to be totally determined, and there are pseudocode instructions which have no machine equivalent but are for bookkeeping and for recording information needed for some debug tables and for object listing control.

A pseudocode instruction occupies 72 bytes of memory and contains the following information:

* Definition-Use chains of instruction results.

* The register assigned to a result, or where it has been spilled to.

* A memory reference chain.

* Either an instruction or an assembler psuedo.

Note that space is allocated to hold all the above information all the time, even when it is not yet needed, such as register assignments during turple translation, or is useful only during the processing of a basic block, such as DU chains. An instruction contains:

* An opcode.

* Up to three register fields.

* Double precision result flag.

* Q-field contents.

* Address group information.

The Q-field is for those pseudocode instructions destined to become a machine instruction which has a Q-field, such as LX or MULXQ. The address group information describes the memory address for those instructions which reference memory, such as LX or SX. A register field may contain either a reference to a previous instruction result or a small constant.

An assembler pseudo contains the following information, only some of it relevant to any given pseudo:

* An opcode.

* Source line number and source line id.

* Statement id and flag indicating a labelled statement.

* Object list on flag.

* Subroutine name being called, index to its argument list, and argument count.

3.3 PSEUDOCODE TRANSLATION

During pseudocode translation, the following are performed:

* Registers are assigned to operands.

* Spill code is generated as necessary.

* Label addresses are determined.

* The number of bytes of generated object code is counted.

Note that the relative offsets for branch instructions are not computed at this time; it is not known yet whether they will all fit in the instruction's Q-field. After translation to pseudocode is complete, the object code size is checked. If greater than 64K, then all branches are converted into their long form, and the label addresses and object code size is adjusted accordingly.

3.4 PREBINARY FORMAT

The prebinary is even closer to machine code. The following types of entries make up a prebinary stream:

* Machine instruction. Ready to be assembled into the code section. Includes a byte count and a tag to be placed in the comment field of the object listing. A CCM record type consisting of packed unaligned bit fields is used to store the instruction. Branches, loads, stores, and calls have their own pseudo.

* Branch pseudo. Special cased because we don't know yet whether the short or long form is needed.

* Beginning of statement. Contains source line number, source line id, statement id, and a statement is labelled flag.

* BSS pseudo. Defines a label or entry point.

* Object list. Turns object listing on or off.

* Call pseudo. The CALLSEG instruction is special cased so that a RIF entry can be created for it. Also, it causes an argument descriptor record to be generated.

* Calltc pseudo. Same as the call pseudo, but for C procedures only.

* LAA psuedo. Loads the address of the read-only section from the binding section. Also creates a RIF entry.

* Load/Store pseudo. Used for all load/store instructions (except when LAA is used). Determines the A register to use and the offset.

* Move bytes pseudo. Specifies source and destination A registers and byte count. The byte count is a constant that's <= 256.

3.5 PREBINARY TRANSLATION

The following needs to be done:

* Most prebinary instructions are simply assembled and put in the loader text record for the code section.

* Branch instructions are fully processed and assembled. If the long form is required, a simple macro expansion is performed to yield the several machine instructions needed.

* RIF loader table entries are generated from loads and calls which access the binding section.

* Debug line table entries are generated.

3.6 LOCAL BLOCK STRUCTURE

The read-only and read-write sections of a program unit are subdivided into numerous local blocks, each holding a different type of data. For most of these blocks, there is a corresponding QCG table which lists its contents. Byte aligned blocks occur before word aligned blocks in a section.

Actual_Arglist	Read/Write	Word aligned
Addresses	Read only	Byte aligned
Arrays	Read/Write	Word aligned
Assigned_Labels	Read only	Word aligned

Char_Arglist	Read only	Word aligned
Char_Constants	Read only	Byte aligned
Char_Ptrs	Read only	Word aligned
Char_Temps	Read/Write	Byte aligned
Characters	Read/Write	Byte aligned
CILIst	Read/Write	Byte aligned
Common_Arglist	Read/Write	Word aligned
Copy_Arg_Control	Read only	Byte aligned
Dimension_Info	Read only	Byte aligned
Dynamic_Addresses	Read/Write	Byte aligned
Dynamic_Char_Ptrs	Read/Write	Word aligned
Encoded_Formats	Read only	Byte aligned
Equivalence	Read/Write	Word aligned
Float_Constants	Read only	Word aligned
Holl_Constants	Read only	Word aligned
Init_Parms	Read only	Word aligned
Int_Constants	Read only	Word aligned
IoList	Read/Write	Byte aligned
Llb_Arglist	Read/Write	Word aligned
Nlst_Array_Descs	Read only	Byte aligned
Nlst_Group_Descs	Read only	Byte aligned
Prog_File_Descs	Read only	Byte aligned
Scalars	Read/Write	Word aligned
String_Descs	Read/Write	Byte aligned
Temps	Read/Write	Word aligned

4.0 FTN/VE V2 QCG DESIGN OVERVIEW

4.0 FTN/VE V2 QCG DESIGN OVERVIEW

4.1 INTERMEDIATE LANGUAGE (IL) FORMAT

IL will replace pseudocode in the new QCG. This is required both for performance and for future OL=MEDIUM support. Unlike pseudocode, IL will be quite simple: an opcode with up to three register fields occupying 16 bytes. A size of 16 bytes is desirable for several reasons. It is large enough to contain the information required by almost all IL instructions but not so large as to be wasteful of memory. It is also a power of 2, which allows CYBIL to generate more efficient subscripting code.

A register field is 4 bytes long and may contain either an I-number (the index of the IL instruction whose result is being used), an address tag, or an index or relative pointer into some table. Which it is is determined by context (i.e., the instruction opcode).

The 4-byte address tag replaces the complex 13-byte psuedo address group of the previous QCG. It contains a 1-byte block id and a 3-byte block offset. For anything other than a symbol or label, the block id specifies a local block and the block offset is a true byte offset. For symbols and labels, the block id is a special tag indicating a symbol or label and the block offset is really an index into the name symbol table or the label symbol table, where the assigned address can be located. This also facilitates optimizations, as loads and stores will refer to a symbol and not to an address.

IL will be more abstract than pseudocode, although it will still maintain a close relationship to Cyber 180 machine instructions. For example, there is no ADDXQ or MULXQ IL instruction. Instead, an ENTER instruction is used to enter the constant value, and an ADDX or MULX instruction will refer to the ENTER instruction. Also, there is just ENTER, no ENTP, ENTN, ENTC, ENTA, ENTE, or even a load from a (single word) constant table. The distinctions between these similar instructions will be made at register allocation and machine code generation time.

4.1.1 BASIC BLOCKS

A basic block starts with either a label definition or follows a branch instruction and terminates either just before another label definition or at a branch instruction. There may be more than one branch in a basic block due to Arithmetic IF and computed GOTO statements; however, they must be the last group of instructions in the block. An unconditional branch, if present, must be the very last instruction. When OL=DEBUG, in addition to the above criteria, a new basic block must be started when end of statement is reached.

I-numbers start from one for each basic block, and an IL instruction may

only reference the I-number of a preceding instruction in the same basic block. The physical layout of IL will permit a basic block to be edited (optimized) without affecting other blocks; i.e., other basic blocks need not be moved or re-numbered, which is what would have to be done in QCG V1.

4.1.2 VECTOR SUPPORT

IL will pretend that there are vector registers, each capable of holding 512 values. There are "load vector register" and "store vector register" instructions, and the I-number of a vector arithmetic operation defines a vector value. During register allocation time, these "vector registers" are mapped onto the array operands themselves and possible temporary storage. Normally, a load/store vector register serves as a bookkeeping device and produces no machine instructions; however, if the stride is not one, then they will be translated into gather/scatter instructions.

4.1.3 CHARACTER SUPPORT

All character assignment and comparisons will be done inline. In QCG V1, none of the comparisons and only the simplest of assignments -- constant substring length .ie. 256 characters -- were done inline. IL instructions to perform these operations are defined.

4.1.4 ALLOCATED ARRAY SUPPORT

Allocated arrays introduce pointers into the language, even if in an "backdoor" fashion. In the IL, allocated arrays will be treated as any other array; i.e., the fact that a pointer must be loaded prior to accessing the array is not explicit. Actually, this really isn't any different from an array that is a dummy argument. The problem of loading addresses is left to register allocation, primarily because almost all addresses are invariant for any instance of execution of a program unit. Even for allocated arrays, the address can change via ALLOCATE or FREE on a (presumably) infrequent basis.

The effect on basic block squeezing is also minimal. Allocated arrays can never interfere with each other or with anything else, as can pointer dereferences in other languages such as CYBIL. An ALLOCATE or FREE statement itself has the same effect as a subscripted store as far as memory interference is concerned.

4.2 TURPLE TRANSLATION

Turples are translated one statement at a time. Because more than one basic block can result from this translation, it is possible that expression results may have to be stored into a temporary variable. To aid in this, as the turple stream is scanned for the next LINE turple -- indicating the beginning of the

next statement -- a reference count is generated so that we know how many times the result of a turple is used by other turples. Only references to expression results are counted. Note that such a turple result can never be referenced by a turple in another statement, which is why processing a statement at a time is convenient. Now, when a new basic block must be started, we know which expression results still have pending references and therefore must be saved.

After the turple references have been counted, the turple stream is rescanned. This time, we look for turples which have side-effects, such as branches, subroutine calls, stores, array conformity checks, substring range checks, but not others like arithmetic operations. When we come across one, first its operands are translated into IL and then the turple itself is translated. Note that turples whose values are never used (and have no side-effects!) are never translated into IL, thereby reducing the need for dead code elimination following basic block squeezing.

This approach is necessary because of vector expressions. It is necessary to see the entire expression in order to analyze its vector semantics; therefore, there's no point in doing anything when we come across an isolated turple which produces only a result and no side-effects. On the other hand, turples such as a store do not produce a result and so should be translated immediately. In addition, their operands will be the root of a complete expression tree, ready to analyze.

Some turples behave in both fashions. ARY_CHK_IX, which specifies an array subscript with subscript checking turned on, has a side-effect and a value. The side-effect is the code to check for a valid subscript and to issue a diagnostic when an error is detected. The value is the actual subscript which is used to compute the word offset of the array element. Turples like this will be processed twice. The first time, it will be processed for its side-effect: its operands will be translated and the error checking code will be generated. The second time, it is processed as a part of the expression tree in which the array element appears as an operand. The operands need not be translated again, of course, and the ARY_CHK_IX turple will be turned into address calculation code, as if it were just a plain ARY_IX turple.

In QCG V1, an expression was always contained in a single basic block. This is no longer true. Inline subscript checking, substring checking, array conformity checking, and loops generated by vector subexpressions can all introduce a basic block boundary in the middle of an expression. When a new basic block is needed, it is necessary to locate all expression values which will be referenced again and save them in temporaries.

Code skeletons are no longer used to translate most turples. The main reason is that the simplistic macro expansion code skeletons simply cannot handle vector semantics, nor the new revised turple structure for representing subscripts. Explicit CYBIL code will now be used. There are other benefits of this as well. First, the macro expansion was a slow process hurting compilation rate. Second, it prohibited many types of peephole optimizations; e.g., MAX(I,0) has a more efficient expansion than MAX(I,J), but there's no way to represent that in a code skeleton.

The following are the basic steps performed during turple translation:

* Variables and arrays are assigned addresses as they are referenced for the first time.

* Complex arithmetic is expanded into real arithmetic.

* Certain intrinsic functions are expanded.

* Basic block boundaries are detected (and the necessary save code produced).

* Basic blocks are squeezed, but only after they are completed.

* Subscripts are expanded into polynomial expressions.

* Numerous tables, such as CILIST, IOLIST, subroutine arguments, constant tables, and others, are built.

* Vector subexpressions are either scalarized or strip-mined.

* Arrays used both as the target of an assignment and as an operand in the same assignment statement are detected and handled.

* Overlapping string assignments are detected and handled (EE=OCM only).

* Certain intrinsic functions which use a CMML routine that can be inserted inline into the object code are identified (except when OL=DEBUG).

4.2.1 VECTOR SUBEXPRESSIONS

Normally, an expression is translated to IL by using simple recursive descent. But when a vector operator is found, we got ourselves a vector subexpression and a different strategy must be adopted.

First, we continue the recursive descent to locate all scalar subexpressions in the vector expressions. As they are found, they are immediately translated to IL. This is necessary for both performance reasons and because you have no choice. Performance is improved because the scalar subexpressions are evaluated prior to any loop that may be created for the vector operators. To prove the second point, consider a scalar subexpression containing a call to a user function!

Now all we have left are the vector operators. We must now decide whether to scalarize or not. Naturally, strip-mining is performed only if the vector lengths may exceed 512 words. We scalarize if any of the following conditions are met:

* Target machine of C180MI has been selected.

* There is an operator which has no vector equivalent in hardware, such as

character concatenation or double precision arithmetic.

* There is an intrinsic function which has no vector expansion nor a vector version in CMML.

* An array section is defined via an array-valued subscript, or any other condition which prohibits strip-mining.

* RANF function is present.

Array-valued intrinsics, such as SUM, and array-valued user functions are handled somewhat differently. The operand is considered to be a separate vector subexpression, which must be completely evaluated prior to function execution, from the subexpression in which the function's value is an operand.

Array-valued intrinsics such as SUM may either be expanded inline or generate a call to a FRTL routine. Generally, these intrinsics can be expanded inline when all dimensions are collapsed and the result is a scalar. Where summing is to be performed only along a single dimension, and especially when that dimension is not known at compilation time, a FRTL routine should be invoked, otherwise the inline expansion would be excessively large.

4.2.1.1 Scalarized_IL_generation

Here's what must be done to scalarize a vector subexpression:

* Create a set of nested loops, with one loop for each dimension and a temporary variable to go with it. (An attempt will be made to collapse adjacent dimensions.)

* Vector operations become their equivalent scalar operations.

* A whole-array operand becomes a subscripted load IL instruction, with the subscript being a function of the loop indexes.

* An array section operand becomes an array element (i.e., a subscripted load IL instruction) by replacing each section reference by a polynomial of the form "lower_bound + loop_index * stride".

* An array-valued subscript becomes a scalar subscript by subscripting the subscript by the loop index.

* The subscript calculations for the assignment target is handled in the same fashion as for array operands.

4.2.1.2 Vector_IL_generation

The steps for generating vector IL instructions are remarkably similar to the steps needed for scalarizing the expression. The differences are as follows:

* The inner-most loop will increment its index variable by 512 instead of 1 for each iteration.

* The inner-most loop can be eliminated entirely if it can be shown it would execute exactly once.

* Vector operations remain vector operations.

* Subscripted loads and stores become subscripted vector loads and stores; i.e., the subscript calculations generated by scalarization are still generated, the difference being that 512 values will be loaded starting from that address instead of only a single value.

4.2.2 BASIC BLOCK SQUEEZING

The purpose of basic block squeezing is to improve both the execution time of a program and the time it takes to compile it by eliminating redundant instructions. An instruction is redundant if it either computes the same result as some previous instruction, or if its effect is nullified by some successor instruction.

Side-effect free instructions whose operands are all I-numbers are the simplest to handle. Two such instructions are equivalent if their operands are identical. When this occurs, the second instruction is deleted and all references to its result are altered to refer to the first instruction's result. This act may uncover more redundant instructions.

A load instruction is redundant if its operands are identical to a previous load instruction AND there is no intervening, interfering store instruction. A load instruction may also be redundant if it references the exact same address as a preceding store instruction. In this case, all references to the load instruction are modified to refer to the value stored by the store instruction. These acts may also uncover more redundant instructions.

A store instruction is redundant if it stores into a scalar variable and there is another store into that variable later on. Stores into an array are never deleted. Note that any load instructions between the two stores are redundant and are deleted, and therefore do not prevent the first store from being redundant.

Equivalenced variables are handled somewhat differently. A store to an equivalenced variable or array is never deleted. Furthermore, such a store will interfere with a load from any other equivalenced variable or array.

Subroutine calls and I/O statements can cause definitions and references of variables that are not normally visible in IL. (They aren't in QCG V1). This is handled by two IL instructions, one which notes the possible reference of a variable and the other which notes a possible definition. These instructions are generated for the arguments of a subroutine call, function call, or I/O statement, when these result in a call to an external routine. It is also assumed that all common variables may be referenced and/or defined by a call to a user subroutine or function.

An ALLOCATE or FREE statement will interfere with any load or store to the

arrays affected.

4.3 IL TRANSLATION

The primary differences between QCG V1 and V2 in how they go about IL translation are as follows:

* Object code size is not computed during prebinary generation.
* Selected CMML routines are expanded and placed inline in the prebinary stream (except when OL=DEBUG).
* Machine instructions are stored in the prebinary stream as raw bytes, ready to be copied into the code section, instead of using CCM's instruction descriptor.

The change in how machine instructions are represented is for performance reasons. The CCM instruction descriptor requires that the instruction be broken into its constituent fields, which are then stored into unaligned packed bit fields. There are other unaligned packed bit fields in the descriptor which specify which fields are present. Prebinary translation then has to reassemble the instruction from these fields. Not fast. There is no reason why the instructions cannot be generated already fully assembled. True, the CCM instruction descriptor must be built when an object listing is being produced, but that is only when the object listing is turned on. Certain types of instructions, such as branches, are still special cased in the prebinary stream, but they too are more assembled than they were before.

4.3.1 INLINE INSERTION OF CMML ROUTINES

A utility will be developed which automatically pulls the loader texts for a CMML routine from the math library and from that generates a set of data declarations that is usable by QCG (and CVCG). See DAP S5267 for the restrictions imposed on CMML that can be inserted inline.

The tables describing the routine to QCG provide the following information:

* The set of registers modified by the routine.
* The size and contents of the code section.
* The size and contents of the read-only data section, if any.
* The size and contents of the binding section, if any, and some indication of which entry points to the read-only data section.

The code section is inserted into the prebinary stream for each call to the

routine. Space is allocated for its data and binding sections the first time the routine is used.

4.4 PREBINARY TRANSLATION

The first step is to count the number of bytes the object code occupies and the number of RIF entries (loader relocation information) required. If the object code size is greater than 64K bytes, some branches may have to be converted to their long form before we can continue.

Prebinary translation is affected the least. Apart from support for vector machine instructions, the major differences are as follows:

* A better algorithm for handling long branches.

* The RIF loader table is preallocated, eliminating a lot of fuss.

* The explicit macro expansion mechanism for pseudo-instructions, such as long branches, has been eliminated, replaced by explicit CYBIL code.

The last two items are for performance reasons.

4.4.1 LONG BRANCHS

The QCG V1 way of handling long branches was to assume that every branch is a long branch if the code section was larger than 64K bytes. QCG V2 attempts to determine which branches are really the long branches. Here's how:

First, the object code is divided into blocks. A label or a branch instruction starts a new block. The addresses of the labels and branches are determined, with the assumption that all branches are long branches. All branches are marked as "long". Long branches are stored as a special pseudo-instruction in the prebinary stream.

Each branch is look at turn. If it's target is less than 64K bytes away, it is turned into a short branh. When all the branches have been examined, the object code size and label addresses are recomputed and we are done.

For OL=MEDIUM, this can be done repetitively until no more branches go from long to short.

5.0 PERFORMANCE CONSIDERATIONS

Minimum criteria for execution and compilation speed are as follows:

OPT	Execution	Compilation
LOW	Faster than or equal to FTN/VE V1.	Faster than or equal to FTN/VE V1.
DEBUG	Faster than or equal to FTN/VE V1.	Faster than or equal to FTN/VE V1.

6.0 REFERENCES

1. "System Interface Standard", S2196.
2. "CYBER 180 CCM Interface Specification", S2987.
3. "CDC STANDARD FORTRAN Language Specification", S4512.
4. "CDC FORTRAN ERS", S4516.
5. "FRTL Interface Specification", S3512.
6. "CYBER 180 Mathematical Library (CMML)", S2929.
7. ANSI X3.9-198x FORTRAN.
8. "CDC FORTRAN General Design Specification", S4228.
9. "CDC FORTRAN Front End General Internal Design", by J. A. Cohen.
10. "Inline insertion of CMML routines", S5267.

Table of Contents

1.0 INTRODUCTION 1-1
1.1 SUMMARY OF REQUIREMENTS 1-1
1.2 OPTIMIZATIONS 1-2

2.0 GLOSSARY 2-1

3.0 FTN/VE V1 QCG DESIGN OVERVIEW 3-1
3.1 TURPLE TRANSLATION 3-1
3.2 PSEUDOCODE FORMAT 3-2
3.3 PSEUDOCODE TRANSLATION 3-3
3.4 PREBINARY FORMAT 3-3
3.5 PREBINARY TRANSLATION 3-4
3.6 LOCAL BLOCK STRUCTURE 3-4

4.0 FTN/VE V2 QCG DESIGN OVERVIEW 4-1
4.1 INTERMEDIATE LANGUAGE (IL) FORMAT 4-1
4.1.1 BASIC BLOCKS 4-1
4.1.2 VECTOR SUPPORT 4-2
4.1.3 CHARACTER SUPPORT 4-2
4.1.4 ALLOCATED ARRAY SUPPORT 4-2
4.2 TURPLE TRANSLATION 4-2
4.2.1 VECTOR SUBEXPRESSIONS 4-4
4.2.1.1 Scalarized IL generation 4-5
4.2.1.2 Vector IL generation 4-5
4.2.2 BASIC BLOCK SQUEEZING 4-6
4.3 IL TRANSLATION 4-7
4.3.1 INLINE INSERTION OF CMML ROUTINES 4-7
4.4 PREBINARY TRANSLATION 4-8
4.4.1 LONG BRANCHS 4-8

5.0 PERFORMANCE CONSIDERATIONS 5-1

6.0 REFERENCES 6-1